Design for Verification
Using Design Patterns to Build Reliable Systems[1]

Peter C. Mehlitz
CSC, NASA Ames Research Center
pcmehlitz@email.arc.nasa.gov

John Penix
NASA Ames Research Center
John.J.Penix@nasa.gov

Abstract

Components are mainly used in commercial software development to reduce time to market. While some effort has been spent on formal aspects of components, most of this was done in the context of integration into programming languages or operating system frameworks. As a consequence, increased reliability of composed systems is merely regarded as a side effect of a more rigid testing of pre-fabricated components.

In contrast to this, Design for Verification (D4V) puts the focus on component specific property guarantees, which are used to design systems with high reliability requirements. D4V components are domain specific design pattern instances with well-defined property guarantees and usage rules, which are suitable for automatic verification. The guaranteed properties are explicitly used to select components according to key system requirements.

The D4V hypothesis is that the same general architecture and design principles leading to good modularity, extensibility and complexity/functionality ratio can be adapted to overcome some of the limitations of conventional reliability assurance methods, such as too large a state space or too many execution paths.

1. Introduction

High dependability systems can be characterized by the need to satisfy a set of key properties at all times. This includes standard properties like absence of deadlocks or constant space execution, and application specific properties such as guaranteed responses or "correct" results. General approaches to demonstrate compliance with these properties are testing and formal verification.

Testing has inherent limitations with respect to non-reproducible execution environment behavior (like thread scheduling), which can be regarded as non-determinisms that are not fully controllable in the test environment. Thus testing covers only a small fraction of the potential state space of concurrent applications.

If higher confidence is needed, formal verification methods, like model checking [1] or static analysis can be used. However, these methods tend to not scale well or suffer imprecision, requiring abstract models to be constructed either manually or with tool support. This process may introduce fidelity problems between the model and the actual design or implementation.

Moreover, creating models can be so expensive that verification becomes a one time effort, which is inconsistent with the evolutionary nature of large systems development. The difficulty to verify formal properties in turn often leads to a lack of properties in the system specification, creating additional fidelity problems by having to guess the verification goals.

To be effective, both testing and verification require a co-operative program design. For testing, design choices mainly determine the achievable test granularity (unit tests). Verification depends on a suitable program design for applicability of its modeling techniques (e.g. for abstraction). This leads to the implication of explicitly using appropriate design measures, instead of compensating their lack by means of tools and modeling techniques.

A general approach to the verification of large systems therefore is to use composition to build a system from separately verifiable parts. This is the approach followed by Design for Verification (D4V), based on the assumption that the same design principles can be used not only to increase verifiability, but also to help testing, and especially to improve understanding and extensibility of the target system.

[1] The research described in this report was performed at NASA Ames Research Center's Automated Software Engineering group and is funded by NASA's Engineering for Complex Systems program.

2. The Design for Verification Approach

In order to improve verifiability, D4V utilizes component selection based on property requirements derived from system specifications.

D4V components are not classical modules. Object oriented designs typically use a mix of inheritance (for static variation) and delegation (for runtime variation). The application mainly provides parts that are hooked into a usually much bigger framework library. The most abstract model for this is not the (language specific) class model, but sets of collaborating types with dedicated roles. This is essentially what came to be known as Design Patterns [2].

Design Patterns are mainly used as "mental building blocks." They come with various degrees of collaboration details, ranging from high level architectural patterns (not explicitly naming interfaces or aggregates) down to language specific idioms (coding patterns at expression level) [7]. Since a primary quality of a design pattern is its genericity, i.e., how readily it can be applied to a range of similar concrete problems, patterns often come with a deliberate lack of formalism, to leave enough freedom for problem-specific implementations. This otherwise helpful simplicity can make it difficult to use automated checks for correct pattern implementation and usage, which is on the other hand required to deduce properties for a target system composed of certain patterns. Bridging this gap between human-oriented fuzziness and tool oriented formalism is the major challenge for the D4V approach.

Ultimately, D4V strives to support the design process at two different levels:

- domain specific pattern systems
- aspect oriented implementation

The first level provides the building blocks from which to compose systems, the second level gives guidelines for how to implement these components.

2.1 Domain Specific Pattern Systems

The D4V pattern systems consist of application domain specific libraries with static pattern instance components, plus a lookup schema to identify suitable patterns.

Each pattern instance comes with a set of guaranteed properties, a set of formal rules how to use the pattern so that the guarantees will hold, and code fragments of the invariant parts of the pattern.

D4V Pattern Instance = Design Pattern Decscription
+ usage rules
+ property guarantees
+ code

Usage rules and property guarantees can be thought of as a generalization of programming-by-contract pre- and post-conditions [6] in the context of a set of collaborating classes with variable, application specific parts.

Usage rules mainly describe expected behavior of application-specific pattern parts, like:

> Implementations of the interface method NonBlockingObserver.update() have to return in bounded time (are not allowed to directly or indirectly block or infinitly loop)

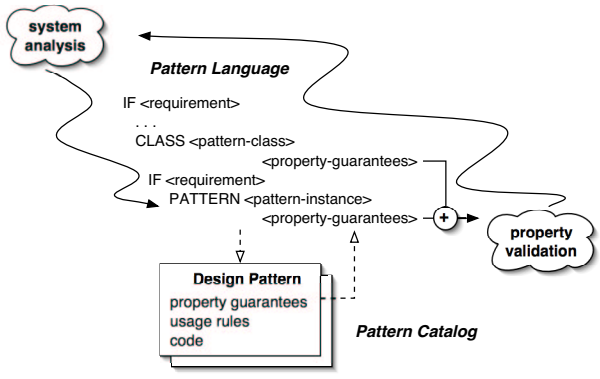
These rules have to be sufficiently formal to support automated checks (e.g. checking for absence of certain library function calls in a given call tree by means of static analysis tools). It is essential to note that these checks are applied to the real target system, thus enabling verification as a development co-process like regression testing.

Property guarantees are derived from the invariant structure of a pattern instance, based on the assumption that usage rules are fulfilled:

> The NonBlockingObserver pattern instance guarantees that Subject notification does not deadlock and all Observers that are registered when the notification is initiated are notified

Code can be either invariant library components, base class and interface (purely behavioral) definitions, or artifacts/skeletons of application specific components.

The lookup schema builds upon approaches from specification-based component retrieval, where components are indexed by formal properties [8], [9]. These indexing methods support both automated retrieval [10], [11] and interactive browsing [12]. To represent the design space defined by the implementation choices made during pattern instantiation, we will build upon Smith's design space representation for algorithms [13], where implementation choices progressively refine an implementation.



The property guarantees form the premier selection criteria for pattern lookup, which constitutes the main principle of D4V: choosing components based on verifiable properties derived from key requirements.

We do not design a system and later on try to find out what properties we can check by means of existing verification tools, but rather design the system based on what we want to verify.

An example for such properties could be an asynchronous event multiplexer (EventQueue) component that guarantees non-blocking, constant time multiplexing and prioritized event retrieval.

--- PATTERN LOOKUP ---

. . .
IF system has async *EventEmitters* and *EventProcessors*
  IF some *EventProcessors* require synchronization
    CLASS *EventMultiplexer*
      EventEmitter not blocked
      EventProcessors sequenced
      IF no *EventEmitter* requires feedback
        IF bounded *ResponseTime* required
          IF closed set of *EventProcessors*
            IF *Events* are prioritized on type
              PATTERN *TypePriorityReactor*
                guaranteed processing order
                bounded response time for
                  sub--critical *EventRate*
                extensible set of EventEmitters
                  and EventProcessors
. . .

--- GLOSSARY ---

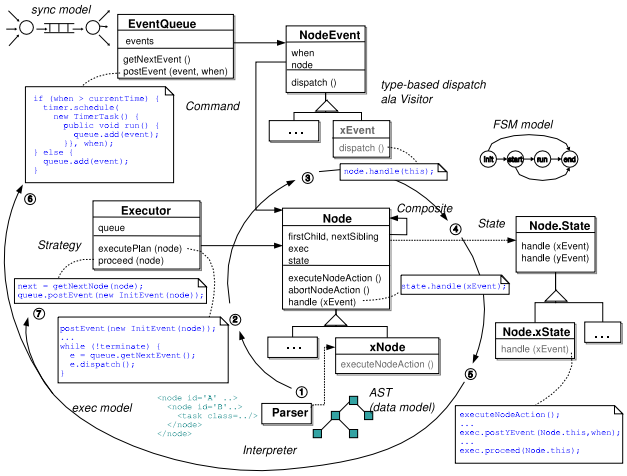
. . .

EventEmitter:	Component where potentially async (environment generated) stimuli enter the system . . . Examples: Sensor, . . .
EventProcessor:	Component which is reacts on the occurrence of Events . . . Examples: Controller, . . .

. . .

While most of the standard design patterns described in [2] can be utilized (i.e. can be annotated with usage rules and property guarantees), D4V does require a certain level of detail for the collaboration of pattern participants. Low level programming language specific idioms and high level architectural patterns are outside the scope of D4V. The initial implementation will focus on variants of well known, standard design patterns (like Observer, Reactor etc.).

Beyond this focus on safe implementation of "essential system complexity," there is also an important side effect of reducing harmful "accidental complexity" [3], which is a typical outcome of adding features to systems that were not designed for extensibility. To quantify this aspect, we have taken a small, moderately object-oriented, autonomous robot application and re designed it using standard design patterns.



The pattern oriented redesign not only resulted in the anticipated extensibility and test-suitability, especially for unit tests, but also showed a significant reduction in overall size, an elimination of complexity "hot spots" (also known as "god classes"[14]), and a reduction of constructs that potentially cause defects (e.g. number of threads).

	Original version	D4V version
classes	82	37
interfaces	1	10
NCLOC	5926	1745
max WMC	397	56
sum WMC	1426	389
threads	6	2

Both systems were written in Java. *Weighted Methods per Class* (WMC) is the sum of the methods' cyclomatic complexities, which is a control flow complexity metric.

2.2 Aspect oriented Implementation

To improve the verifiability of designs, D4V focuses on three essential aspects [4] of its pattern implementations, each one being represented by explicitly marked and annotated code sections:

- consistent program states (CheckPoints)
- conceptual process model (ControlPoints)
- potential extensions (ExtensionPoints)

The intention behind these aspects is to unify the inherent design model(s) and the implementation of a system in a way that these models are made explicit, are

consistently preserved, and later-on can be easily extracted for automated verification purposes.

(a) ***CheckPoints*** are locations where necessarily consistent, property relevant states have to be checked. This includes freely placeable assertions as well as pre-, post-conditions and invariants. CheckPoints describe the correctness model of a component implementation.

The checks themselves can refer to explicit program state (variable values) and implicit execution environment state (number of instructions, relative time etc.).

While CheckPoints can be used for non-functional properties (like progress, resource consumption etc.), they are mandatory for the verification of functional, i.e. application specific properties, and are linked to their corresponding pattern guarantees.

If the programming environment has a assertion mechanism, evaluation of CheckPoints is straight forward. Reachability analysis and side-effect detection of check points should be supported by specialized tools.

To ease integration with object oriented design (especially inheritance-aware pre-, post-conditions and invariants), the implementation of CheckPoints should preferably use languages supporting Programming-by-Contract [6]. Alternatively, CheckPoints can be implemented by means of comment based pre-processors (like *iContract* [15]).

An example could be a set of functional properties to be fulfilled by a event dispatcher component used to process a “plan” consisting of a collection of action nodes:

```
--- PROPERTY GUARANTEES ---
P1.1 : no overlapping plan execution ...
P1.2 : upon plan execution start, all nodes
       are in InitState ...
P1.3 : upon plan execution termination, all
       nodes are in EndState ...
P1.4 : after plan got processed, all node
       resources held by plan nodes are freed
```

```
void processPlan () {
  /** @pre: [P1.1]
      verifier.isNull(startNode);
      Handle snap =
         verifier.getResourceSnapshot();
  */

  startNode = readPlan();
  /** @check: [P1.2]
      verifier.allNodesInInitState(..);
  */

  startProcessing(startNode);
  while (!terminate) {
    Event e = queue.getNextEvent();
    e.dispatch();
  }

  /** @check: [P1.3]
      verifier.allNodesInEndState(..);
  */

  releasePlan(startNode);

  /** @post: [P1.4]
      verifier.allResourcesFreed(snap);
   */
}
...
```

Pre- and post-conditions are kept separate from checks (free assertions) because they have to comply with inheritance rules (overridden functions have to accept at least base pre-conditions, and have to guarantee at least base post-conditions).

There are several potential levels of tool support. Property guarantees can be checked for missing CheckPoints. Test runs can be checked against coverage of CheckPoints. A target system based model checker like *JPF* [1] can be used to check for assertion violations. The above example uses a 'verifier' delegation object to enable such a model checker to perform the checks outside of the applications state space (to avoid increasing the state space by also verifying the checks themselves).

(b) ***ControlPoints*** describe the process model of the system, denoting locations that are relevant for both testing and model checking. Only conceptual ControlPoints are identified, not every branch in the control flow. This especially includes potentially context switching operations in multi-threaded programs (like Thread start, locking attempts etc.), reflecting the observation that

- concurrent programs should be built around their major synchronization points (e.g. message dispatching loops)
- many concurrency defects (like deadlocks, missed signals, nested monitor lockout etc.) can be detected by analyzing these synchronization points, i.e. do not require fine grained interleaving at statement or instruction level

The minimal information associated with each ControlPoint is its identifier (including thread/process), the liveness-relevant type of operation (wait, lock etc., including ControlPoints this operation depends on), and the set of follow on ControlPoints.

```
class Emitter implements Runnable {
  ...
  void run () {
    ...
    /** @cp: emitter.post */
    queue.postEvent(..);
  }
  ...
}

class Processor implements Runnable {
  ...
  void run () {
    /** @cp: dispatcher.enter */
    ...
```

```
    /** @cp: .loop next(.get,.exit) */
    while (!terminate) {
      ...
      /** @cp: .get wait(emitter.post)
                    next(.loop) */
      Event e = queue.getNextEvent();
      ...
    }

    /** @cp: .exit */
  }
  ...
}
...
```

For testing, ControlPoints can be used as a basis for required coverage, including (automated) test case generation.

For model checking, they can be considered as the "built-in model" providing potential backtracking targets and atomic sections, to avoid the state space explosion that comes with fine-grained interleaving. The ControlPoint information could be especially useful to give the model checker hints that are suitable to implement search heuristics (e.g. to perform a "depends first" scheduling heuristic to detect missed signal defects).

Describing the process model outside the implementation language itself makes it possible to extend the level of detail so that it can be also used for less complex tools than a program model checker (e.g. searching for potentially harmful patterns in ControlPoint sequences), but it comes with the downside that ControlPoint specifications are redundant with the program itself.

To avoid this redundancy, we are also investigating program designs using explicit ControlPoint APIs, especially choice generator objects in branches. By encapsulating the choice selection mechanism in a replaceable component, it becomes possible to switch between different verification strategies (e.g. to use "depends-first" thread scheduling to provoke missed signal defects).

A welcome side effect of this approach would be that systems can be tested and verified in their real target environment, instead of the environment which is required to run complex tools.

The result would be a hybrid model checking approach similar to the state-less model checking of Verisoft [16].

(c) ***ExtensionPoints*** describe the underlying extension model of a design by identifying the relevant concepts and constructs.

The reason why we focus on this aspect in the D4V context is the fact that the development of a complex system is hardly ever completed [5]. The typical case is a evolutionary extension of functionality, which can easily lead to accidental complexity and feature bloat (un-controlled accumulation of functions), violating properties which did hold in the original implementation.

While it is not feasible nor desired to reflect every possible extension in the systems design, there should be provisions for the anticipated major extensions in terms of identifying the

- relevant base classes together with their overridable methods
- delegation types, object and configuration methods
- generic types with corresponding type parameter constraints

ExtensionPoints are mainly a measure of documenting these constructs so that (a) new feature requests can be easily assessed regarding their feasibility, and (b) completed extensions can be checked against the original design.

ExtensionPoints are closely related to UML class diagrams, but can span several classes, or can combine several extension aspects in a single class. A typical case is marking overridable primitive operations used in a TemplateMethod pattern [2].

There are two general categories of extensions: mandatory and optional. Mandatory extensions include application specific concrete types to create instances of patterns using abstract delegation types. Optional extensions mainly include subclasses to create pattern variants.

The following example shows extension points of a robotics application executing a plan consisting of user provided primitive operations.

--- EXTENSION MODEL ---

...

ActionType: optional extension

Abstraction for entities consisting of application specific action objects ("what to do" *ActionInstance*) and plan specific conditions ("when to do it") ...

ActionInstance: mandatory parameter

Abstraction for application provided primitive operations...

```
/** @ext: ActionType subclass */
class Node {
  /** @ext: ActionInstance delegate */
  Action action;

  /** @ext: ActionInstance call */
  Node (Action act, ..) {..}
  ...
  /** @ext: ActionType override */
  protected void executeNodeAction()
  {..}
  ...
}

/** @ext: ActionInstance implement */
interface Action {..}
```

With these annotations, the application specific parts to implement a certain design can be quickly identified,

and subsequent extensions can be checked for design-compatible subclassing.

ExtensionPoints are not meant to be used as a replacement for the language specific type system (ensuring existence of required methods and other subclassing constraints) or correctness related semantics (using inheritance-aware CheckPoints).

3. Project Status and Outlook

The D4V project is still in its early stages. The current focus is on the development of a suitable design pattern system based on our motivating example, a event driven, observable, state-model based control system for autonomous robots. We plan to eventually have three different versions of the system as a basis for metrics comparison

- the original version exposing typical effects of accidental complexity
- the standard design-pattern implementation to show the reduction of complexity and increase of extensibility
- a version that uses D4V specific patterns to show the property-guarantee driven design process

This approach reflects our view that D4V is not a radically new design methodology, but rather extends and combines already accepted "best design practices" in order to overcome the traditional gap between design/development and testing/verification, which causes not only problems for finding defects, but also for subsequently fixing them.

References

[1] W. Visser, K. Havelund, G. Brat, S. Park. "Model Checking Programs", Proceedings of the 15th International Conference on Automated Software Engineering (ASE), Grenoble, France, September 2000.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - "Design Patterns Elements of Reusable Object-Oriented Software", Addison Wesley, 1995

[3] Frederick P. Brooks - "No Silver Bullet: Essence and Accidents of Software Engineering", Proceedings of the IFIP '86 conference

[4] Tzilla Elrad, Robert Filman, Atef Bader - " Aspect Oriented Programming", CACM Vol 44 No. 10, October 2001

[5] David Parnas - "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, SE-5(2):128--38, Mar. 1979.

[6] Bertrand Meyer - "Object Oriented Software Construction", . Prentice Hall 1997

[7] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern-Oriented Software Architecture : A System Of Patterns", Wiley, 1996

[8] Amy Moormann Zaremski, Jeannette M. Wing. "Specification matching of software components" 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995

[9] A. Mili, R. Mili, and R. Mittermeir, "Storing and retrieving software components: A refinement based system", IEEE Transactions on Software Engineering, 23(7):445-460, July 1997

[10] John Penix and Perry Alexander "Efficient Specification-Based Component Retrieval", Automated Software Engineering, vol. 6, pp. 139-170, Kluwer Academic Publishers, April 1999

[11] B. Fischer, J. Schumann, G. Snelting, "Deduction-Based Software Component Retrieval". Automated Deduction - A basis for applications, Vol. III: Applications, W. Bibel and P. H. Schmitt, eds., pp. 265-292, Kluwer 1998.

[12] B. Fischer "Specification-Based Browsing of Software Component Libraries", Journal of Automated Software Engineering, Vol. 7, No. 2, 2000, pp. 179-200

[13] Douglas R. Smith "Toward a Classification Approach to Design", Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96, LNCS 1101, Springer Verlag, 1996

[14] Arthur J. Riel, "Object-Oriented Design Heuristics", Addison-Wesley, 1996

[15] Reto Kramer "iContract - the Java Design by Contract tool" , Proceedings of Technology for Object-Oriented Languages and Systems, TOOLS-USA. IEEE Press, 1998

[16] Patrice Godefroid,"Model checking for programming languages using Verisoft", Symposium on Prinicples of Programming Languages, pages 174-186. ACM, 1997